module 2 :

Complexité algorithmique

Comment comparer les performances de deux algorithmes effectuant les mêmes tâches ? Deux paramètres essentiels (mais pas uniques) sont à prendre en compte : le temps d'exécution, et l'espace mémoire requis. En outre, cette comparaison doit être indépendante de l'ordinateur utilisé, et sur un même ordinateur du langage de programmation utilisé pour traduire cet algorithme[1]. Il s'agit donc de rendre la comparaison indépendante de tout aspect technique conjoncturel.

• Complexité spatiale

Il fut un temps où la quantité de mémoire utilisée était le critère principal, de par le caractère excessivement onéreux de la mémoire des ordinateurs. Ce temps est révolu, et de plus les nouveaux modes de programmation (programmation fonctionnelle en particulier) tendent à masquer ce coût. Nous n'évoquerons donc que brièvement cet aspect dans le chapitre consacré à la notion de récursivité.

• Complexité temporelle

Pour rendre l'étude du coût temporel d'un algorithme indépendante de tout aspect physique, il faut prendre pour unité de mesure une opération élémentaire pertinente pour le problème étudié. Calculer le *coût*, ou la *complexité*, d'un algorithme, c'est alors déterminer le nombre d'opérations élémentaires effectuées par cet algorithme. Avant toute chose, il conviendra donc de s'entendre sur la notion d'opération élémentaire : affectation, comparaison entre nombres ou caractères, opérations algébriques, ...

Par exemple, pour comparer deux algorithmes effectuant essentiellement des calculs numériques, on dénombrera les opérations coûteuses que sont les multiplications, les racines carrées, les logarithmes,...A contrario, la comparaison entre deux algorithmes de tri se fera par le biais du dénombrement des accès aux emplacements de mémoire et des comparaisons entre données.

• Outillage mathématique

De nombreux algorithmes dépendent d'un paramètre entier n : le nombre n d'entiers de Hamming qu'on cherche à déterminer, le taille n d'un tableau qu'on cherche à trier, ..., et le calcul du nombre exact d'opérations élémentaires en fonction de n n'est pas toujours réalisable. De plus, comparer deux algorithmes n'a d'intérêt que pour de grandes valeurs de n ; on cherchera donc avant tout un *équivalent* de ce nombre plutôt que le nombre exact.

Même ainsi, ce calcul n'est pas toujours possible. Dans la pratique, on se contentera souvent d'utiliser les notations de Landau pour mesurer un coût.

Rappel. On note : $u_n = O(v_n)$ lorsqu'il existe une constante $k_1 > 0$ telle que :

$$\forall n \in \mathbb{N},\ u_n \leqslant k_1 v_n.$$

Intuitivement, cet notation traduit une majoration du coût : dire que $u_n = O(n^2)$, c'est dire que u_n va croître *au plus* proportionnellement au carré de n (mais sa croissance peut être en fait linéaire). De manière implicite, les informaticiens utilisent cette notation pour mesurer le pire des cas : dire que $u_n = O(n^2)$ ne veut pas dire que le coût sera toujours quadratique, mais que dans le pire des cas, il le sera.

Pour traduire plus précisément l'idée que deux suites ont le *même ordre de grandeur*, on introduit une nouvelle notation :

Notation. On note : $u_n = \Theta(v_n)$ lorsqu'il existe deux constantes $k_1 > 0$ et $k_2 > 0$ telles que :

$$\forall n \in \mathbb{N},\ k_1 v_n \leqslant u_n \leqslant k_2 v_n.$$

On peut alors classer les algorithmes suivant leur complexité c_n, qui sera qualifiée de :
- *logarithmique* lorsque $c_n = \Theta(\log n)$;
- *linéaire* lorsque $c_n = \Theta(n)$;
- *quasi-linéaire* lorsque $c_n = \Theta(n \log n)$;
- *quadratique* lorsque $c_n = \Theta(n^2)$;
- *polynomiale* lorsque $c_n = \Theta(n^k)$ avec $k > 0$;
- *exponentielle* lorsqu'il existe $\lambda > 1$ tel que $\lambda^n = O\big(c_n\big)$.

[1] Le temps d'exécution d'un algorithme écrit dans un langage donné peut même dépendre du compilateur utilisé pour traduire ce programme en langage machine.

1. Coût minimal d'un algorithme de tri

Pour illustrer ce module, nous allons montrer comment la notion d'*arbre de décision* permet de minorer le coût d'un algorithme de tri.

La notion d'*arbre binaire* est définie récursivement ; il s'agit :
- soit d'une *feuille* ;
- soit d'un *nœud* d'ou partent deux flèches dirigées vers deux sous-arbres.

Il s'agit d'une structure de donnée ; à ce titre, elle contient de l'information, stockée dans les feuilles et les nœuds (ce sont les *étiquettes*) ainsi que dans les flèches (les *labels*).

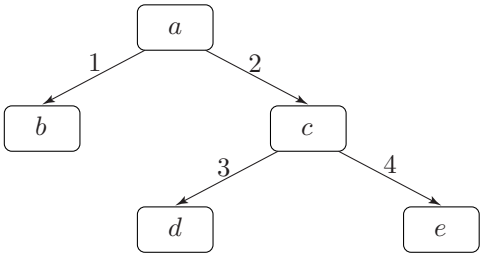

FIG. 1 – un exemple d'arbre binaire

L'arbre représenté figure 1 est formé de deux nœuds (étiquetés par les lettres a et c) et trois feuilles (étiquetées b, d et e). le nœud étiqueté a est la *racine* de l'arbre ; le sous-arbre pointé par la flèche de label 1 est son *fils gauche* ; celui pointé par la flèche de label 2 son *fils droit*.

On définit la *hauteur* d'un arbre binaire comme suit : la hauteur d'un arbre réduit à une feuille est 0 et la hauteur d'un arbre non réduit à une feuille est égal à 1 plus le maximum des sous-arbres fils de sa racine.

Q1 Quel est le nombre maximal de feuilles d'un arbre binaire de hauteur $h \in \mathbb{N}$?

- **Arbres de décision**

Soit $n \in \mathbb{N}^*$ un entier ; on désire faire un choix parmi n possibilités données dans un ensemble $\mathcal{C}$ de cardinal n. Un *arbre de décision* est un arbre binaire tel que :
- la racine est étiquetée par $\mathcal{C}$;
- chacune des feuilles est étiquetée par un singleton inclus dans $\mathcal{C}$;
- chaque nœud qui n'est pas une feuille est étiqueté par un sous-ensemble $\mathcal{C}_1$ de $\mathcal{C}$ et ses deux fils par des sous-ensembles stricts et non vides $\mathcal{C}_2$ et $\mathcal{C}_3$ de $\mathcal{C}_1$ vérifiant : $\mathcal{C}_1 = \mathcal{C}_2 \cup \mathcal{C}_3$.

De plus, on adjoint à chaque nœud un test booléen ; les deux flèches issues de ce nœud portent respectivement les labels v (pour *vrai*) en direction de $\mathcal{C}_2$, et f (pour *faux*) en direction de $\mathcal{C}_3$.

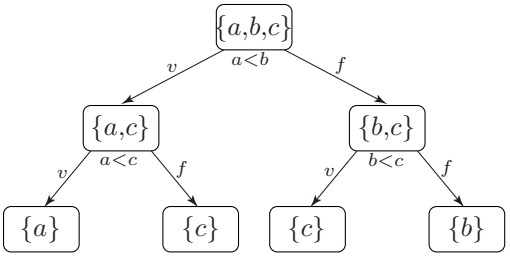

FIG. 2 – un exemple d'arbre de décision

L'instruction définie par un nœud d'un arbre de décision est la suivante : on sait que le choix à réaliser se trouve dans l'ensemble $\mathcal{C}_1$; on effectue le test correspondant ; si le test est vrai le choix est à réaliser dans $\mathcal{C}_2$; sinon il est à réaliser dans $\mathcal{C}_3$.

Exemple. L'arbre dessiné figure 2 est un arbre de décision permettant de choisir le plus petit parmi trois nombres a, b, c.

Q2 **a)** Quelle est la hauteur minimale d'un arbre de décision pour un choix dans un ensemble de cardinal n ?

b) Quelle est la hauteur maximale d'un arbre de décision pour un choix dans un ensemble de cardinal n ?

- **Algorithmes de tri**

On se pose maintenant le problème du tri d'un tableau t contenant n éléments.

Q3 **a)** Le tableau résultat du tri est une permutation de ce tableau. Combien existe-t-il de permutations du tableau t si les éléments contenus dans t sont deux à deux distincts ?

b) On peut considérer le problème du tri du tableau t comme un algorithme de choix parmi les permutations de t et dont les tests sont des comparaisons entre deux éléments du tableau. Donner la valeur de la hauteur minimale de l'arbre de décision correspondant à ce choix.

c) En déduire que la meilleure complexité envisageable d'un algorithme de choix pour le tri d'un tableau est un $\Theta(n \log n)$.

Remarque. Nous verrons plus tard qu'il existe effectivement de tels algorithmes.

2. Exercices

1 – Vous êtes face à un mur qui s'étend à l'infini dans les deux directions. Il y a une porte dans ce mur, mais vous ne connaissez ni la distance, ni la direction dans laquelle elle se trouve. Par ailleurs, l'obscurité vous empêche de voir la porte à moins d'être juste devant elle.

Décrire un algorithme vous permettant de trouver cette porte en un temps linéaire vis-à-vis de la distance qui vous sépare de celle-ci.

2 – Supposez que l'on vous donne n pièces d'or. Vous savez que parmi elles se trouve une seule fausse pièce, qui ne se distingue des autres que par son poids (mais vous ignorez si elle est plus lourde ou plus légère que les autres). Votre problème est de la trouver en un minimum de pesées.

- Dans le cas où $n = 3^p$, montrer que si l'on sait si la fausse pièce est plus lourde ou plus légère, p pesées suffisent à la déterminer.
- Dans le cas général, montrer que si l'on sait si la fausse pièce est plus lourde ou plus légère, $\lceil \log_3 n \rceil$ pesées suffisent à la déterminer.

Dorénavant, on ne suppose plus que l'on sait si la fausse pièce est plus lourde ou plus légère que les autres.

- Dans le cas où $n = 3^p$, montrer que $p + 1$ pesées suffisent.
- En déduire que dans le cas général, $\lceil \log_3 n \rceil + 1$ pesées suffisent.

Remarque : $\lceil x \rceil$ désigne l'unique entier $p \in \mathbb{Z}$ tel que $p - 1 < x \leqslant p$, et $\log_3 x$ le logarithme en base 3 de x. Ainsi, $\lceil \log_3 n \rceil$ désigne l'unique entier p tel que $3^{p-1} < n \leqslant 3^p$.

■

Réponses

Q1 On montre par récurrence sur $h \in \mathbb{N}$ qu'un arbre de hauteur h possède au plus 2^h feuilles.

– C'est clair lorsque $h = 0$.

– Si $h \geqslant 1$, on suppose le résultat acquis au rang $h-1$. Les deux fils de la racine d'un arbre de hauteur h possèdent donc au plus 2^{h-1} feuilles, soit 2^h feuilles en tout.

Q2 **a)** Un arbre de décision pour un choix dans un ensemble de cardinal n doit posséder au moins n feuilles ; sa hauteur h vérifie donc : $2^h \geqslant n$, soit : $h \geqslant \lceil \log_2 n \rceil$.

b) Démontrons par récurrence sur k qu'un nœud de niveau k est étiqueté par un ensemble non vide d'au plus $(n-k)$ éléments.

– Lorsque $k = 0$, c'est clair puisque la racine est étiquetée par $\mathcal{C}$, de cardinal n.

– Lorsque $k > 0$, supposons les nœuds de niveau $k-1$ étiquetés par des ensembles non vides d'au plus $n-k+1$ éléments. Notons $\mathcal{C}_1$ un de ceux-ci. Si $\mathcal{C}_1$ n'est pas l'étiquette d'une feuille, ses deux fils sont étiquetés par des sous-ensembles stricts et non vides $\mathcal{C}_2$ et $\mathcal{C}_3$, dont le cardinal est compris entre 1 et $n-k$.

De ceci il résulte que les feuilles sont au plus au niveau $n-1$; c'est donc la hauteur maximale d'un arbre de décision.

Q3 **a)** Il existe bien évidemment $n!$ permutations différentes.

b) Un arbre de décision pour le tri d'un tableau doit donc, d'après la question 2.a, avoir une hauteur au minimum égale à $\lceil \log_2(n!) \rceil = \left\lceil \sum_{k=2}^{n} \log_2 k \right\rceil$.

c) La croissance de la fonction ln permet l'encadrement : $\displaystyle\int_{k-1}^{k} \ln t \, \mathrm{d}t \leqslant \ln k \leqslant \int_{k}^{k+1} \ln t \, \mathrm{d}t$ et donc :

$$\int_{1}^{n} \ln t \, \mathrm{d}t \leqslant \ln(n!) \leqslant \int_{2}^{n+1} \ln t \, \mathrm{d}t \iff n \ln n - n + 1 \leqslant \ln(n!) \leqslant (n+1)\ln(n+1) - 2(\ln 2 - 1).$$

On en déduit : $\ln(n!) \underset{+\infty}{\sim} n \ln n$ puis : $\lceil \log_2(n!) \rceil \underset{+\infty}{\sim} n \log_2 n$.

Le coût d'un algorithme de tri (en nombre de comparaisons) est égal à la hauteur de l'arbre de décision qui lui est associé ; d'après ce qui précède, on ne peut espérer mieux qu'un $\Theta(n \log n)$.

■